3.1) Создание Бинарной кучи на Java

import java.util.ArrayList;

import java.util.List;

public class BinaryHeap {

private List<Integer> heap;

public BinaryHeap() {

heap = new ArrayList<>();

}

private int parent(int i) { return (i-1)/2; }

private int left(int i) { return 2\*i + 1; }

private int right(int i) { return 2\*i + 2; }

public void insert(int key) {

heap.add(key);

siftUp(heap.size()-1);

}

public Integer extractMin() {

if (heap.isEmpty()) return null;

int minVal = heap.get(0);

heap.set(0, heap.get(heap.size()-1));

heap.remove(heap.size()-1);

if (!heap.isEmpty()) {

siftDown(0);

}

return minVal;

}

private void siftUp(int i) {

while (i > 0 && heap.get(parent(i)) > heap.get(i)) {

swap(parent(i), i);

i = parent(i);

}

}

private void siftDown(int i) {

int minIndex = i;

int left = left(i);

int right = right(i);

if (left < heap.size() && heap.get(left) < heap.get(minIndex)) {

minIndex = left;

}

if (right < heap.size() && heap.get(right) < heap.get(minIndex)) {

minIndex = right;

}

if (i != minIndex) {

swap(i, minIndex);

siftDown(minIndex);

}

}

private void swap(int i, int j) {

int temp = heap.get(i);

heap.set(i, heap.get(j));

heap.set(j, temp);

}

public boolean isEmpty() {

return heap.isEmpty();

}

}

3.2) Создание Биноминальной кучи на Java

class BinomialNode {

int key;

int degree;

BinomialNode parent;

BinomialNode child;

BinomialNode sibling;

public BinomialNode(int key) {

this.key = key;

this.degree = 0;

this.parent = null;

this.child = null;

this.sibling = null;

}

}

public class BinomialHeap {

private BinomialNode head;

public void insert(int key) {

BinomialHeap newHeap = new BinomialHeap();

newHeap.head = new BinomialNode(key);

merge(newHeap);

}

public void merge(BinomialHeap other) {

head = mergeLists(head, other.head);

other.head = null;

if (head == null) return;

BinomialNode prev = null;

BinomialNode curr = head;

BinomialNode next = curr.sibling;

while (next != null) {

if (curr.degree != next.degree ||

(next.sibling != null && next.sibling.degree == curr.degree)) {

prev = curr;

curr = next;

} else if (curr.key <= next.key) {

curr.sibling = next.sibling;

linkTrees(next, curr);

} else {

if (prev == null) {

head = next;

} else {

prev.sibling = next;

}

linkTrees(curr, next);

curr = next;

}

next = curr.sibling;

}

}

private BinomialNode mergeLists(BinomialNode h1, BinomialNode h2) {

if (h1 == null) return h2;

if (h2 == null) return h1;

BinomialNode newHead;

BinomialNode current;

if (h1.degree <= h2.degree) {

newHead = h1;

h1 = h1.sibling;

} else {

newHead = h2;

h2 = h2.sibling;

}

current = newHead;

while (h1 != null && h2 != null) {

if (h1.degree <= h2.degree) {

current.sibling = h1;

h1 = h1.sibling;

} else {

current.sibling = h2;

h2 = h2.sibling;

}

current = current.sibling;

}

if (h1 != null) {

current.sibling = h1;

} else {

current.sibling = h2;

}

return newHead;

}

private void linkTrees(BinomialNode child, BinomialNode parent) {

child.parent = parent;

child.sibling = parent.child;

parent.child = child;

parent.degree++;

}

}

3.3) Создание кучи Фибонччи на Java

class FibonacciNode {

int key;

int degree;

boolean marked;

FibonacciNode parent;

FibonacciNode child;

FibonacciNode left;

FibonacciNode right;

public FibonacciNode(int key) {

this.key = key;

this.degree = 0;

this.marked = false;

this.parent = null;

this.child = null;

this.left = this;

this.right = this;

}

}

public class FibonacciHeap {

private FibonacciNode minNode;

private int nodeCount;

public FibonacciHeap() {

minNode = null;

nodeCount = 0;

}

public void insert(int key) {

FibonacciNode node = new FibonacciNode(key);

if (minNode == null) {

minNode = node;

} else {

addToRootList(node);

if (key < minNode.key) {

minNode = node;

}

}

nodeCount++;

}

private void addToRootList(FibonacciNode node) {

node.left = minNode;

node.right = minNode.right;

minNode.right.left = node;

minNode.right = node;

}

public Integer extractMin() {

if (minNode == null) return null;

FibonacciNode z = minNode;

// Добавляем детей в корневой список

if (z.child != null) {

FibonacciNode child = z.child;

do {

FibonacciNode nextChild = child.right;

addToRootList(child);

child.parent = null;

child = nextChild;

} while (child != z.child);

}

// Удаляем z из корневого списка

z.left.right = z.right;

z.right.left = z.left;

if (z == z.right) {

minNode = null;

} else {

minNode = z.right;

consolidate();

}

nodeCount--;

return z.key;

}

private void consolidate() {

int maxDegree = (int) Math.floor(Math.log(nodeCount) / Math.log(2)) + 1;

FibonacciNode[] degreeTable = new FibonacciNode[maxDegree];

FibonacciNode current = minNode;

java.util.ArrayList<FibonacciNode> nodes = new java.util.ArrayList<>();

// Собираем все корневые узлы

do {

nodes.add(current);

current = current.right;

} while (current != minNode);

for (FibonacciNode node : nodes) {

int degree = node.degree;

while (degreeTable[degree] != null) {

FibonacciNode other = degreeTable[degree];

if (node.key > other.key) {

FibonacciNode temp = node;

node = other;

other = temp;

}

link(other, node);

degreeTable[degree] = null;

degree++;

}

degreeTable[degree] = node;

}

// Восстанавливаем minNode

minNode = null;

for (FibonacciNode node : degreeTable) {

if (node != null) {

if (minNode == null || node.key < minNode.key) {

minNode = node;

}

}

}

}

private void link(FibonacciNode child, FibonacciNode parent) {

// Удаляем child из корневого списка

child.left.right = child.right;

child.right.left = child.left;

// Добавляем child как дочерний для parent

child.parent = parent;

if (parent.child == null) {

parent.child = child;

child.left = child;

child.right = child;

} else {

child.left = parent.child;

child.right = parent.child.right;

parent.child.right.left = child;

parent.child.right = child;

}

parent.degree++;

child.marked = false;

}

}

3.4)Создание Хеш таблицы на java

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.List;

public class HashTable<K, V> {

private List<LinkedList<Entry<K, V>>> buckets;

private int capacity;

private int size;

private double loadFactor;

private static class Entry<K, V> {

K key;

V value;

Entry(K key, V value) {

this.key = key;

this.value = value;

}

}

public HashTable() {

this(8, 0.75);

}

public HashTable(int capacity, double loadFactor) {

this.capacity = capacity;

this.loadFactor = loadFactor;

this.size = 0;

initializeBuckets();

}

private void initializeBuckets() {

buckets = new ArrayList<>(capacity);

for (int i = 0; i < capacity; i++) {

buckets.add(new LinkedList<>());

}

}

private int hash(K key) {

return Math.abs(key.hashCode()) % capacity;

}

public void put(K key, V value) {

if ((double) size / capacity >= loadFactor) {

resize();

}

int index = hash(key);

LinkedList<Entry<K, V>> bucket = buckets.get(index);

for (Entry<K, V> entry : bucket) {

if (entry.key.equals(key)) {

entry.value = value;

return;

}

}

bucket.add(new Entry<>(key, value));

size++;

}

public V get(K key) {

int index = hash(key);

LinkedList<Entry<K, V>> bucket = buckets.get(index);

for (Entry<K, V> entry : bucket) {

if (entry.key.equals(key)) {

return entry.value;

}

}

return null;

}

private void resize() {

int newCapacity = capacity \* 2;

List<LinkedList<Entry<K, V>>> oldBuckets = buckets;

capacity = newCapacity;

initializeBuckets();

size = 0;

for (LinkedList<Entry<K, V>> bucket : oldBuckets) {

for (Entry<K, V> entry : bucket) {

put(entry.key, entry.value);

}

}

}

public boolean containsKey(K key) {

return get(key) != null;

}

public int size() {

return size;

}

}